Minimum number of trials to reach from source word to destination word

# Minimum number of coins to make change

Given an infinite supply of coins of values: {C1, C2, ..., Cn} and a sum. Find minimum number of coins that can represent the sum.

Consider values set as {2, 5, 3}, n = length of values = 3  
and sum = 7  
Then we can make change for 7 by reducing the given coin values one by one and finding if we can make of the remaining amount.  
For example, for 7, we subtract 2, 5, 3 one by one and then find out if we can make change of the remaining amounts:  
1-> (7-2) = 5,  
2-> (7-5) = 2,  
3-> (7-3) = 4 respectively.  
Of all the possibilities we find the one which gives us minimum number of coins i.e. minimum number of coins from #1, #2 and #3 above and add 1 to it.  
So, the next step in this problem is to find out minimum number of coins to make change for 5, 2, 4 which can be found by applying same strategy as above taking sum as 5, 2 and 4 and finally stopping when no further amount can be reduced from sum or sum becomes 0.  
If no further values can be reduced from sum to make change, and it is still non zero, then this path does not lead to a solution.  
  
Hence, we have following recurrence relation:  
    If sum = 0, minCoins = 0 - because no coins are required to make change for 0.  
    else minCoins(sum) = min(minCoins(sum-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
    If there is no i for which sum >= value[i], then minCoins(sum) = ∞ which is represented by Integer.MAX\_VALUE  
  
For finding minimum number of coins for sum = 7:  
A) minCoins(7)   
               = min(minCoins(7-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(7-2), minCoins(7-5), minCoins(7-3)) + 1  
               = min(minCoins(5), minCoins(2), minCoins(4)) + 1  
  
B) minCoins(5)   
               = min(minCoins(5-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(5-2), minCoins(5-5), minCoins(5-3)) + 1  
               = min(minCoins(3), minCoins(0), minCoins(2)) + 1  
  
C) minCoins(3)   
               = min(minCoins(3-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(3-2), minCoins(3-3)) + 1,             because 3 < 5, we do not consider it  
               = min(minCoins(1), minCoins(0)) + 1  
  
D) minCoins(1) = ∞    because there is no value in set {2, 5, 3} which is less than 1 so 1 cannot be represented using any of these coins.  
  
E) minCoins(0) = 0  
  
Therefore,   
C) minCoins(3)   
               = min(minCoins(1), minCoins(0)) + 1 = min(0, ∞) + 1 = 1  
  
F) minCoins(2)   
               = min(minCoins(2-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(2-2)) + 1  
               = min(minCoins(0)) + 1  
               = 1  
  
Therefore,   
B) minCoins(5)   
               = min(minCoins(3), minCoins(0), minCoins(2)) + 1 = min(1, 0, 1) + 1 = 0 + 1 = 1  
  
G) minCoins(4)   
               = min(minCoins(4-values[i])) + 1, for all i = 0 to n-1, where sum >= values[i]  
               = min(minCoins(4-2), minCoins(4-3)) + 1  
               = min(minCoins(2), minCoins(1)) + 1  
               = min(1, ∞) + 1  
               = 1 + 1  
               = 2  
  
Finally,   
A) minCoins(7)   
               = min(minCoins(5), minCoins(2), minCoins(4)) + 1  
               = min(1, 1, 2) + 1  
               = 2  
  
Time complexity of recursive algorithm is exponential.  
An implementation of this algorithm is provided in the code snippet section.  
  
To solve this problem in a more efficient way, we can use Dynamic Programming.  
As, we can see from above example, that we are recalculating the solution for sub problems again and again. For example, we calculate minCoins(2) again in the sub problems minCoins(4) and minCoins(5).  
If we create an array for already calculated values, then it will save us a lot of recomputation.  
So, we create a minCoins array - minCoins[sum+1] where minCoins[i] represents minimum number of coins required to make change for amount = i.  
We build up the array in bottom up manner starting with minCoins[0].  
So, for any j, if we want to find minCoins[j] then minCoins[0...j-1] will already have been calculated and we just need to find out minimum of minCoins[j-values[i]] where i = 0...n-1 and j >= values[i]  
  
The time complexity of the Dynamic Programming solution is O(n\*sum).  
The space complexity is O(sum).  
Here n is length of the array of values.